Prof. Dr. A. Poetzsch-Heffter
Mathias Weber, M.Sc.

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2015)

Practice Sheet 6

Date of Issue: 26.05.15
Deadline: 02.06.15
(before the lecture as PDF via E-Mail)

Exercise 1 Immutability

Discuss for the following Java classes whether or not they are immutable with respect to the definitions of immutability given in the lecture.

a)
```
public final class ImmList {
    public static final ImmList EMPTY = new ImmList(0, null);

    private final int head;
    private final ImmList tail;

    private ImmList(int head, ImmList tail) {
        this.head = head;
        this.tail = tail;
    }

    public ImmList cons(int i) {
        return new ImmList(i, this);
    }

    public int head() {
        if(isEmpty()) {
            throw new UnsupportedOperationException("head␣of␣empty␣list");
        }
        return this.head;
    }

    public ImmList tail() {
        if(isEmpty()) {
            throw new UnsupportedOperationException("tail␣of␣empty␣list");
        }
        return this.tail;
    }

    public boolean isEmpty() {
        return this == EMPTY;
    }

    public int length() {
        ImmList it = this;
        int len = 0;
        while(!it.isEmpty()) {
            len += 1;
            it = it.tail();
        }
        return len;
    }
}
```

b)
```
public final class CompactString {
    private char[] data;
    private int start;
    private int end;
    private int hash = -1;

    public CompactString(String s) {
        this.data = s.toCharArray();
        this.start = 0;
        this.end = s.length() - 1;
    }

    private CompactString(char[] data, int start, int end) {
        this.data = data;
        this.start = start;
        this.end = end;
    }

    public CompactString substring(int start, int end) {
        if(start < 0 || this.start + end > this.end) {
            throw new IllegalArgumentException("Out of bounds");
        }
        return new CompactString(this.data, this.start + start, this.start + end);
    }

    public void compact() {
        char[] newData = new char[end - start + 1];
        for(int i=start; i<=end; i++) {
            newData[i-start] = data[i];
        }
        this.data = newData;
        this.start = 0;
        this.end = newData.length - 1;
        this.hash = -1;
    }

    @Override
    public int hashCode() {
        if(this.hash >= 0) {
            return this.hash;
        } else {
            int newhash = 0;
            for(int i=start; i<=end; i++) {
                newhash += i ^ data[i];
            }
            this.hash = newhash;
            return newhash;
        }
    }

    @Override
    public boolean equals(Object o) {
        if(o == null || !(o instanceof CompactString)) {
            return false;
        }
        return this.toString().equals(((CompactString)o).toString());
    }

    @Override
    public String toString() {
        return new String(data, start, end - start + 1);
    }
}
```

Exercise 2 Confined Types

Examine the code, available with this practice sheet on the web, with respect to confinedness. The classes `ProofTreeNodeIt` und `ProofContainer` should provide the externally visible interfaces and are thus not confined.

a) Examine the class `ConfinedList`. Can we declare this class as confined? Can we modify the implementation to make it confined?

b) Examine the class `ProofTreeNode` and answer the same questions as above.

c) Examine the class `PTNIterator` and answer the same questions as above.

d) Download and install the tool "kacheck" from the lecture homepage. This tool is able to check the confinedness of Java classes based on the bytecode of these classes.

   Compile Java source files mentioned in this exercise using the java compiler. Afterwards run the command `kacheck -violations` on the folder the class files are in.

   Compare the output of the tool with the results you expected. Try to explain unexpected results.

   Hint: The tool outputs violations for all classes of the Java runtime that are indirectly used. These outputs can be ignored.

Exercise 3 Alias Modes

Consider the class `ConfinedList` from the exercise above. The array stored in the field `data` should be part of the representation of a `ConfinedList`-object, i.e. we annotate it with `rep`.

```
class ConfinedList<T> extends AbstractList<T> implements List<T>, RandomAccess, Cloneable, Serializable {
...
 private /*rep*/ transient Object[] data;
 ...
}
```

a) Is it possible to annotate the remaining parts of the class `ConfinedList`, such that it conforms to the programming discipline presented in the lecture (slide 4.27)? If not, what is the problem with the implementation?

b) Which properties can the programming discipline guarantee? Compare them with the guarantees given by confined types.